In this assignment, we are supposed to take input from the user, check its validity in terms of syntax and implement the given commands if it is possible.

In order to that, we needed to parse the input first. Hence, we constructed a BNF grammar design according to provided language rules. Our recursive syntax grammar looks like this:

```
<Input> ::= <Sentences>
          | <Question>
          | exit

<Question> ::= <Subjects> total <Item> ?
             | <Subject> where ?
             | Who at <Location> ?
             | <Subject> total ?

<Sentences> ::= <Sentence> and <Sentences>
              | <Sentence>

<Sentence> ::= <Actions>
             | <Actions> if <Conditions>

<Actions> ::= <Action> and <Actions>
            | <Action>

<Action> ::= <Subjects> buy <Items>
           | <Subjects> buy <Items> from <Subject>
           | <Subjects> sell <Items>
           | <Subjects> sell <Items> to <Subject>
           | <Subjects> go to <Location>

<Conditions> ::= <Condition> and <Conditions>
               | <Condition>

<Condition> ::= <Subjects> at <Location>
              | <Subjects> has <Items>
              | <Subjects> has less than <Items>
              | <Subjects> has more than <Items>.

<Subjects> ::= <Subject> and <Subjects>
             | <Subject>

<Items> ::= <Quantifier> <Item> and <Items>
          | <Quantifier> <Item>

<Item> ::= ___

<Subject> ::= ___

<Location> ::= ___
```

After that, the coding part started. To tokenize the input, we have
written 2 functions.
*One of those was to look at the next token without moving forward along
the input.
*The other one is the actual tokenizer, since it sets the current_token
variable and input pointer position to the next one.
Those functions gets the next word of the input, categorizes it to
keywords or other things accordingly.

Here is a part of the function mentioned above:

```
/*
void getNextToken() {
    while (isspace(*input)) ++input;
    if (*input == '\0' || *input == '\n') {
        current_token = TOKEN_END;
    } else if (strncmp(input, "exit ", 5) == 0 || strncmp(input,
"exit\0", 5) == 0 || strncmp(input, "exit\n", 5) == 0) {
        current_token = TOKEN_EXIT;
        input += 4;
    }
    ...
*/
```

Then, for each block in the BNF Grammar, we have written a function which
returns 1 if it is syntatically correct (0 otherwise).
At the top, we call Sentences() and it calls Sentence(). In case of
success (1) and a following "and" token, it calls Sentences()
recursively.
Same logic applies for each block, such that every function call others.
In case of a possible sequence, it calls itself recursively.

Here is two example of those blocks:

```
/*
int Subjects(int type) {
    if (Subject(type) == 1) {
        if (peekToken() == TOKEN_AND) {
            getNextToken();
            return Subjects(type);
        } else {
            return 1;
        }
    }

    else {
        return 0;
    }
}
*/


/*
```

```
int Condition() {
    if (Subjects(1) == 1) {
        conditionSubjectNameNo = 0;
        if (peekToken() == TOKEN_AT) {
            sentences[sentenceNo].conditions[conditionNo].keyword =
TOKEN_AT;
            getNextToken();
            return Location(1);
        }

        else if (peekToken() == TOKEN_HAS) {
            sentences[sentenceNo].conditions[conditionNo].keyword =
TOKEN_HAS;
            getNextToken();
            if (peekToken() == TOKEN_LESS) {
                getNextToken();
                if (peekToken() == TOKEN_THAN) {
                    sentences[sentenceNo].conditions[conditionNo].keyword
= TOKEN_HAS + TOKEN_LESS;
                    getNextToken();
                    return Items(1);
                } else {
                    return 0;
                }
            }
...
*/
```

Since a Condition is formed by a Subjects - has/at - Items/Location sequence, it checks the existence of those blocks respectively.

As one may notice while reading the Conditions() function, those functions also set some values to some arrays or attributes.
It is actually for the next part of the assignment, in which we must understand what to do from the input.
Before even calling parsing functions, we created some structs for a Sentence, Item, Action etc. Similar to BNF design, those structs have attributes of their own.

For example:

```
/*
typedef struct {
    SubjectNameStruct subjectNames[MAX_SUBJECT_NAMES];
    int keyword;
    ItemStruct items[MAX_ITEMS];
    char *location;
} ConditionStruct;
/*
```

A Condition have some Subjects, a condition type (has/-more/-less/at) denoted by keyword, Items list or a Location.

We have defined the attributes and initialize them to 0 before parsing starts.
While we check the correctness of syntax, we also fill those attributes.
If the syntax is invalid, then we do not need to use those attributes, so it is not a big deal whether those are filled even if they shouldn't be.
To determine which subject belongs which condition or which condition belongs which sentence, we used array indexes. We increment the indexes everytime a call is succesfully returned 1.

For example:

```
/*
int Conditions() {
    if (Condition() == 1) {
        conditionNo++;
        conditionSubjectNameNo = 0;
        conditionItemsNo = 0;
...
*/
```

Here we have a new Condition in a Conditions sequence. Hence we increment the condition number. Also we set the condition's attributes to zero, since it is a new one.

One of the biggest challenges we were faced during this process, was to differentiate sequences since they are all seperated by same token, "and".

"Ahmet buy 1 bread if Mehmet at Turkiye *and* Samet go to Istanbul"
and may be the seperator of sentences or conditions. To make the program able to understand the difference, we needed to check a step further if we encounter an "and" token.

For instance:

```
/*

        char *backup = input;
        int backup_token = current_token;   // saving our current input
pointer position

        if (peekToken() == TOKEN_AND) {
            getNextToken();
            if (peekToken() != TOKEN_QUANTIFIER) {  // checking whether
the upcoming token is an item
                input = backup;
                current_token = backup_token;                     // if
not, reload the original pointer position and return 1
                return 1;
            } else {
                return Items(type);                     // if so, keep
checking other items in this sequence.
            }
        }

*/
```

This is a part taken from Items() function. If Item() returns 1, which means an Item is found, we look for a potential sequence situation. With the code above, we handle "and" token correctly, whether it is the seperator of Items or Conditions or Actions.

The last part is implementing the commands or questions but before that, we check if there is any duplicates which causes invalidity.

To impelement a sentence command, we first check conditions one by one. If conditions are met, we manipulate the arrays accordingly (executeActions()).

Here is a part of executing actions:

```
/*
void executeActions(SentenceStruct sentence){

    for (int i = 0; sentence.actions[i].keyword != 0; ++i) {
        ActionStruct action = sentence.actions[i];
        if (action.keyword == TOKEN_BUY){
            for (int j = 0; action.subjectNames[j].subjectName != NULL;
++j) {
                for (int k = 0; action.items[k].identifier != NULL ; ++k)
{
                    subjectBuyItem(action.subjectNames[j].subjectName,
action.items[k].identifier, action.items[k].quantifier);
                }
            }
        }else if (action.keyword == TOKEN_FROM){
            subjectsBuyItemsFrom(action);

...
*/
```

Same logic applies for other actions, too.

At the end, we take input from user until the exit command. To do that we used a simple while loop and fgets() function:

```
while (true){

        input = (char *) malloc(max_length * sizeof(char));
        if (input == NULL) {
            printf("malloc error");
            exit(EXIT_FAILURE);
        }

        printf(">> ");
        fflush(stdout);
        if (fgets(input, max_length, stdin) != NULL){
            if (parseExit() == 1) {
                break;
            }else{
                parseAll();
            }
        }
    }
```

Thus, we accomplished the assignment of taking input, interpret and implement it.